(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 July 2005 (07.07.2005)

PCT

(10) International Publication Number
WO 2005/062170 A2

(51) International Patent Classification[7]: G06F 9/45

(21) International Application Number:
PCT/IB2004/052600

(22) International Filing Date:
30 November 2004 (30.11.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
03104774.9 18 December 2003 (18.12.2003) EP

(71) Applicant *(for all designated States except US)*: **KONINKLIJKE PHILIPS ELECTRONICS N.V.** [NL/NL]; Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **HOOGENDIJK, Paul, F.** [NL/NL]; c/o Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL). **HOOGERBRUGGE, Jan** [NL/NL]; c/o Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL). **PAUW, Jurjen, P.** [NL/NL]; c/o Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL).

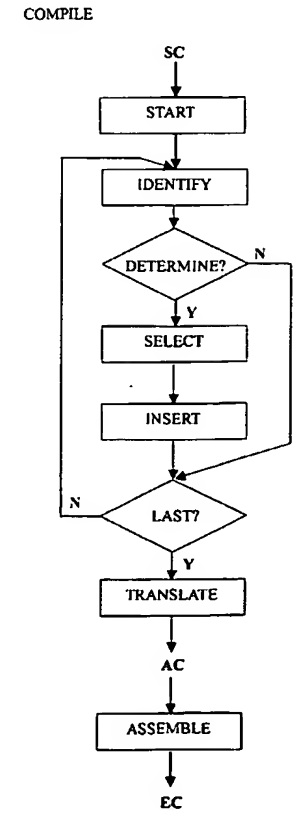
(74) Agents: **ELEVELD, Koop, J.** et al.; Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,

[Continued on next page]

(54) Title: METHOD AND COMPILATION SYSTEM FOR TRANSLATING SOURCE CODE INTO EXECUTABLE CODE



(57) Abstract: The invention relates to a method for translating source code into executable code. The invention also relates to a compilation system arranged to translate source code into executable code, and to a computer program product comprising a program, the program being conceived to be compiled by such a compilation system. The invention relies on the perception that the choice of implementing mutual exclusiveness of a critical section by means of additional instructions, i.e. the determination whether additional instructions are necessary and the selection of appropriate additional instructions, must be left to the compilation system instead of the software developer. Depending on the situation, the compilation system can choose the correct implementation of mutual exclusiveness of the critical section and, if necessary, select appropriate additional instructions and insert them into the source code.

WO 2005/062170 A2

FR, GB, GR, HU, IE, IS, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii)) for the following designations AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW, ARIPO patent (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG)*

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

Method and compilation system for translating source code into executable code

The invention relates to a method for translating source code into executable code. The invention also relates to a compilation system arranged to translate source code into executable code, and to a computer program product comprising a program, the program being conceived to be compiled by such a compilation system.

There are many programming languages available for the software developer, such as Fortran, C, C++, Java etc. All of these so-called third generation programming languages have in common that they can be read by humans, as opposed to executable code which can only be read by a computer. The latter is often referred to as machine-readable code or as executable code. Typically, a program that should be executed on a computer is first translated into assembly code, which is a lower level programming language containing basic instructions for the computer. A so-called assembler then translates these instructions into executable code, which can be executed by the computer's processor. Compilers are deployed for translating source code into executable code. There are many compilers on the market, varying from generic compilers for a variety of programming languages (for example the standard C compiler on the UNIX operating system) to more dedicated compilers which are specific for a certain processor or a special operating system. It is noted that each compiler may use its own version of the assembly language; the translation from assembly code into executable code is a part of the compilation process. There are also compilers which do not use the assembly language; these compilers use a specific internal representation of the source code before translating it into executable code.

Usually it is desirable that the source code can be compiled to as many types of computers as possible without having to be changed. In other words, the source code must be 'portable'.

A program usually modifies variables in the source code of the program. A multi-threaded or multi-tasking data processing system is capable of executing several programs in a simultaneous, pseudo-parallel manner. The processor is then equipped with a scheduler, which is able to switch between programs in order to obtain better resource

utilization or in order to meet real-time deadlines, for example. The scheduler can bring a program temporarily to a halt while allowing another program to start or to resume its execution, and after a while the halted program can resume its execution. A multi-processor system can execute different programs in a truly parallel manner, since each processor is capable of executing one of the programs. The programs are often referred to as tasks, processes or threads. Switching between programs is also referred to as task switching.

In a multi-threaded system, it is possible that several tasks attempt to read and/or modify the same variables. In that case the variables are referred to as global or shared variables. It is noted that the concept of global or shared variables can be broadened to include shared resources, for example I/O devices. In that case, it is possible that several tasks attempt to access the same I/O device. If access to shared or global variables by several tasks is possible without any precaution, then there is a high risk of data corruption. Interference is defined as a situation wherein multiple tasks attempt to read or modify the same variables or the same set of variables at the same time. There are two causes of interference:

- task switching performed by a scheduler in a single-processor system or in a multi-processor system;
- the execution of parallel tasks in a multi-processor system.

For example, a first task may attempt to read data, while the data are being changed by a second task, although it may not have been the programmer's intention. Using the incorrectly read data for further calculations would then lead to incorrect behavior. In order to avoid this kind of data corruption, a concept known as critical sections can be used. A critical section is defined as a piece of source code which contains accesses to global or shared variables that may not be accessed by multiple tasks simultaneously. The purpose of defining a critical section is to avoid undesired interference. Interference may also be desired, which means that it is the programmer's intention that multiple tasks access the same variables.

When defining a critical section, the programmer typically adds explicit instructions to the source code which disable interrupts, for example. If no interrupts can occur during execution of the critical section, then it is not possible to switch from one task to another and no interference can occur in a single-processor system. This is an example of mutual exclusiveness of the critical section. In general, mutual exclusiveness is achieved for a task executing a critical section if other tasks which can access the same variables are brought to a halt at the moment they try to access the variables. The mutual exclusiveness of

the critical section can be implemented by adding explicit instructions to the source code or by using known library routines.

The additional instruction(s) or the library routines can perform a variety of functions, for example clearing and restoring interrupts, acquiring and releasing semaphores and stopping all schedulers. The selection of a function is typically performed by the programmer; the selection depends among others on the number of processors in the system and on the probability that a task switch will occur. On the one hand, if the programmer is not sure about the said probability, he will usually define a critical section as a precautionary measure although it may not be necessary. On the other hand, the programmer may erroneously assume that a task switch cannot occur and not define a critical section, in which case there is the risk of data corruption. The programmer does not have compilers at his disposal to fully support his programming practice with regard to the implementation of mutual exclusiveness of critical sections.

The library routines must be called upon explicitly in the source code. Typically, the routine is a precompiled piece of code residing in the library. A declaration of functions comprised in the routine is generally placed in a header file which is included in the source code of the program; the declaration of a function is referred to as a prototype in C programs. At compile time, the compiler then links the library code to the code of the program so that the declared functions are available.

For example, in a C program a header file called "mutex.h" would be included at the top of a source file to make available a plurality of routines relating to the implementation of mutual exclusiveness. Since the routine(s) add a piece of code to the program, the executable code is longer and slower in terms of execution time. The same occurs if the programmer inserts explicit additional instructions into the source code. This is no problem if the routine or the additional instructions implementing mutual exclusiveness of a critical section are really necessary for a certain piece of source code. However, it is possible that the routine or additional instructions are only inserted in the source code as a precautionary measure. For example, additional code for implementing mutual exclusiveness has been inserted into the program although no task switching will occur and no parallel tasks will execute during execution of that critical section. Known compiling methods and compilers are unable to fully assist the software developer in determining whether it is necessary to insert specific routines or additional code implementing mutual exclusiveness of critical sections. This leads to an inefficient programming practice. It may also lead to non-portable source code.

It is an object of the invention to provide a method and a compilation system assisting the software developer in creating an efficient program which comprises correct implementations of mutual exclusiveness of critical sections. This is achieved by providing a method for translating source code into executable code, characterized by the characterizing portion of claim 1. It is also achieved by providing a compilation system arranged to translate source code into executable code, characterized by the characterizing portion of claim 11.

The invention relies on the perception that the choice of implementing mutual exclusiveness of a critical section by means of additional instructions, i.e. the determination whether additional instructions are necessary and the selection of appropriate additional instructions, must be left to the compilation system instead of the software developer. Depending on the situation, the compilation system can choose the correct implementation of mutual exclusiveness of the critical section and, if necessary, select appropriate additional instructions and insert them into the source code. The additional instructions are sometimes referred to as synchronization primitives. The compilation system may comprise one or more compilers, the compilers being arranged to compile different programs and parts of a program (modules). The compilation system is able to carry out the methods of compilation as defined in claims 1 up to and including 10.

The additional instructions can be in the form of explicit instructions, as defined in claim 2. The additional instructions can also be in the form of one or more library routines, as defined in claim 3. It would even be possible to implement the mutual exclusiveness by means of a combination of explicit instructions and library routines, if appropriate. Typically, the routines reside in a library as precompiled pieces of code, which is called upon by the program. The explicit instruction(s) are sometimes referred to as inline code. The inline code implements the required mutual exclusiveness of the critical section and there is no need for a call to a library routine, which is advantageous if the additional code is relatively small and a call to a library routine would only slow down the program.

In some cases, in particular if no task switching can occur and if no parallel tasks can be executed, no additional instructions are needed at all, as defined in claim 4. If task switching can occur and/or parallel tasks can be performed, it is possible that the critical section is part of a task which does not interfere with other tasks that can be active at the same time: also in that case no additional instructions are needed, as defined in claim 5.

Claims 6, 7, 8, 9 and 10 provide a number of examples of the implementation of mutual exclusiveness of critical sections. In order of appearance, the following alternatives are given:

- disabling and restoring interrupts;
- acquiring and releasing semaphores;
- stopping and restarting schedulers of interfering tasks;
- stopping and restarting all schedulers in a system;
- stopping and restarting at least one processor in a system.

An embodiment of the compilation system is defined in claim 12, wherein the beginning and the end of a critical section are identified by predefined statements in the source code. Such predefined statements can be used advantageously in a program comprised in a computer program product, the program being conceived to be compiled by the compilation system, as claimed in claim 13. Such a program contains the definition of a critical section; the implementation of mutual exclusiveness of the critical section is then performed by the compilation system.

The present invention is described in more detail with reference to the drawings, in which:

Fig. 1 illustrates a known method for translating source code into executable code;

Fig. 2 illustrates a method for translating source code into executable code according to the invention;

Fig. 3A illustrates an example of an additional instruction for implementing mutual exclusiveness of a critical section;

Fig. 3B illustrates another example of an additional instruction for implementing mutual exclusiveness of a critical section;

Fig. 3C illustrates yet another example of an additional instruction for implementing mutual exclusiveness of a critical section;

Fig. 3D illustrates a further example of an additional instruction for implementing mutual exclusiveness of a critical section;

Fig. 3E illustrates a yet further example of an additional instruction for implementing mutual exclusiveness of a critical section;

Fig. 4A and Fig. 4B illustrate an example of interfering tasks;

Fig. 4C and Fig. 4D illustrate how critical sections are defined according to the invention.

Fig. 1 illustrates a known method COMPILE for translating source code SC into executable code EC. The source code SC is typically written in a high level programming language, for example C, C++, Java, Fortran or Pascal. Because these languages are well understood by humans, they can be used to specify the desired behavior of the computer in abstract and logical instructions rather than in machine-specific instructions. After the compilation process starts START, the source code SC is translated TRANSLATE into assembly code AC. The assembly code AC is a lower level programming language which can be used to specify basic instructions for the machine. An assembler then assembles ASSEMBLE the assembly code AC to produce the executable code EC, which can be executed directly by a computer's processor. Alternatively, a compiler may convert the source code to an internal representation of the source code instead of to assembly code. The internal representation of the source code is then translated into executable code. Another possibility is that the compiler may use an internal representation of the source code which is translated into assembly code; subsequently the assembly code is assembled to produce the executable code.

The major disadvantage of this known method is that the choice for implementing mutual exclusiveness of a critical section of the source code is always left to the programmer. The programmer can then adopt different programming styles. In a single-processor system, he may decide that the mutual exclusiveness need not be implemented with additional instructions because he expects that no task switching will occur during execution of the critical section. This makes the resulting program more efficient. However, there is a high risk of data corruption because task switching may occur despite of the programmer's expectations, and certainly on another processor the behavior would be very unpredictable because the scheduler(s) may function in another way. In other words, the resulting program is not portable. On the other hand, the programmer may decide to implement mutual exclusiveness of all critical sections as a precautionary measure. This secures the data, but it is an inefficient programming practice because the resulting program is significantly larger and slower.

Fig. 2 illustrates a method COMPILE for translating source code SC into executable code EC according to the invention. After the compilation process starts START,

the beginning and the end of a critical section of the source code SC are identified IDENTIFY. For example, this can be done by identifying predefined statements such as 'BCS()' (Begin of Critical Section) and 'ECS()' (End of Critical Section) at the beginning respectively at the end of the critical section. The programmer merely inserts these statements into the source code and leaves the implementation of mutual exclusiveness to the compilation system. Subsequently, it is determined DETERMINE? whether an additional instruction (or several additional instructions) is needed to implement mutual exclusiveness of the critical section. The conclusion of this step may be that no additional instruction is needed; the mutual exclusiveness is already achieved because no task switching may occur and no parallel tasks can be executed. In other words, the critical section may be translated into assembly code AC without addition of synchronization primitives.

If needed, one or more appropriate additional instructions are selected SELECT to implement mutual exclusiveness of the critical section. The selected additional instruction(s) is/are then inserted INSERT into the source code SC such that the mutual exclusiveness is implemented in a correct way. The steps of identifying IDENTIFY up to and including inserting INSERTING are repeated for all critical sections of the source code. If the mutual exclusiveness of the last critical section has been implemented, the result of the step LAST? is 'Y' and the translation process can start. Again, the source code (SC) is translated TRANSLATE into assembly code AC and an assembler assembles ASSEMBLE the assembly code AC to produce the executable code EC.

If the compilation system determines that no undesired interference can occur during execution of the critical section, then it is not necessary to insert an additional instruction into the source code. For example, for a single processor wherein only jump instructions can be interrupted (such as a TriMedia processor), this is the case if it is certain that the critical section does not contain jump instructions, or if the compilation system is capable of translating the critical section into assembly code which doesn't contain jump instructions. In this case, no additional instruction is necessary to implement mutual exclusiveness of this critical section.

In other cases the critical section may contain jump instructions; these instructions introduce the uncertainty that undesired interference may occur. However, depending on the instruction set, special jump instructions are available which cannot be interrupted. If this is the case then the compiler may substitute all normal jump instructions within the critical section with these special jump instructions. For the single processor described above (wherein only jump instructions can be interrupted) this measure is sufficient

to guarantee that no undesired interference will occur during execution of the critical section. The additional instruction then merely consists of substituting the jump instructions.

Undesired interference can occur as a result of task switching in a single-processor system, and as a result of task switching or parallel tasks in a multi-processor system. The compilation system can be arranged such that it performs an inspection of the source code, and such that it derives from the source code inspection whether only one task or multiple tasks can be active simultaneously. If only one task can be active at a time, then the solution is straightforward: a task cannot be interrupted and no additional instruction is needed. If multiple tasks can be active, then the compilation system may be further arranged to derive from the source code inspection whether undesired interference can occur. If it is certain that no undesired interference can occur, then again no additional instruction is needed to implement mutual exclusiveness of the critical section. If undesired interference can occur, then the compilation system can select at least one appropriate additional instruction for implementing mutual exclusiveness of the critical section and insert it into the source code.

Fig. 3A illustrates an example of an additional instruction INSTRUCTION for implementing mutual exclusiveness of a critical section, which is suitable for avoiding undesired interference in a single-processor system, effectively disallowing a scheduler to switch tasks. The additional instruction INSTRUCTION comprises:

- an instruction to save the interrupt flag SAVE_INT at the beginning of the critical section;
- an instruction to disable interrupts DISABLE_INT at the beginning of the critical section;
- an instruction to restore the interrupt flag RESTORE_INT at the end of the critical section.

Typically, the interrupt flag can have two values: 'ENABLE' and 'DISABLE'. If the interrupt flag has the value 'ENABLE', then the scheduler allows that task switching occurs during execution of a piece of code. If the interrupt flag has the value 'DISABLE', then the scheduler does not allow that task switching occurs during execution of a piece of code.

The instruction to disable interrupts DISABLE_INT sets the interrupt flag to the value 'DISABLE', thereby ensuring that no task switching can occur during execution of the critical section. Before performing this instruction, the value of the interrupt flag must be saved; the interrupt flag may already have the value 'DISABLE' because critical sections can

be nested. After execution of the critical section, the interrupt flag can be reset to its saved value.

Fig. 3B illustrates another example of an additional instruction INSTRUCTION for implementing mutual exclusiveness of a critical section. This additional instruction can advantageously be used in a multi-processor system to avoid undesired interference. The semaphore mechanism provides for a 'token' that can be claimed by a task; if the token is taken by a task then other tasks cannot use the same token until it has been released. Tasks which access and manipulate the same data can share such a token to guarantee that they do not disturb each other's operations. The token can best be compared with a special variable shared by interfering tasks, the value of which is set to 'TAKEN' if one of the interfering tasks starts executing, thereby signaling the remaining interfering tasks that they cannot start executing. The additional instruction INSTRUCTION comprises:

- an instruction to acquire a semaphore ACQUIRE_SEM at the beginning of the critical section;
- an instruction to release the semaphore RELEASE_SEM at the end of the critical section.

The relationship between the affected global or shared variables and the available semaphores is predetermined. Because the critical section may access several global or shared variables, it is possible that more than one semaphore should be acquired. In that case, the instruction to acquire a semaphore ACQUIRE_SEM should be repeated until all relevant semaphores have been acquired. The skilled person will recognize that precautions must be taken in the case of multiple semaphores such that deadlock is prevented, for example by correctly ordering of the instructions to acquire the semaphores. The semaphores must be released after execution of the critical section, so the instruction to release the semaphore RELEASE_SEM should then also be repeated.

It is noted that semaphores are suitable when interfering task have a common semaphore, in which case disabling interrupts would be a too drastic measure. The use of semaphores then provides an efficient alternative. The programmer may indicate which alternative should be used to implement mutual exclusiveness of the critical section, for example by setting a parameter of the compilation system indicating which alternative must be used for all critical sections in a program and/or by providing the function BSC() with a parameter indicating which alternative must be used for a specific critical section. Preferably the selection of the most efficient alternative can be left to the compilation system, so that the programmer does not need to perform an additional analysis of the program.

Fig. 3C illustrates yet another example of an additional instruction INSTRUCTION for implementing mutual exclusiveness of a critical section. This solution can be used advantageously either in a single-processor or in a multi-processor system. However, it must be possible to determine for which tasks the undesired interference occurs. This will be a subset of all tasks. The preferred way to determine this is through an inspection of the source code performed by the compilation system. If this is possible, then it suffices to stop the schedulers corresponding to those tasks, so that task switching is disabled only for the tasks which interfere. The additional instruction INSTRUCTION comprises:

- an instruction to register REGISTER which schedulers are running at the beginning of the critical section;
- an instruction to stop schedulers of interfering tasks STOP_SCH at the beginning of the critical section;
- an instruction to start the schedulers of interfering tasks START_SCH which were running, at the end of the critical section.

Fig. 3D illustrates a further example of an additional instruction INSTRUCTION for implementing mutual exclusiveness of a critical section. The additional instruction INSTRUCTION stops all schedulers; it can be used when it is too difficult to determine for which subset of tasks the undesired interference occurs. Stopping all schedulers ensures that task switching does not occur on each processor in a system. Although in this way it is ensured that no task switching occurs during execution of the critical section, it can be an expensive way to achieve mutual exclusiveness. The additional instruction INSTRUCTION comprises:

- an instruction to register REGISTER which schedulers are running at the beginning of the critical section;
- an instruction to stop all schedulers STOP_ALLSCH at the beginning of the critical section;
- an instruction to restart the schedulers START_ALLSCH which were running, at the end of the critical section.

Fig. 3E illustrates a yet further example of an additional instruction INSTRUCTION for implementing mutual exclusiveness of a critical section. It can be deployed in a multi-processor system. It is possible to shut down one or more processors in order to avoid the execution of interfering tasks. This is also an expensive solution, but it may necessary. The solution may be more advantageous than stopping all schedulers, for example if a first task executing on a first processor interferes with a number of second tasks, all of

which execute on a second processor. In that case, it is not recommended to stop all schedulers in the system, but rather to shut down the second processor while the first task executes the critical section. The additional instruction INSTRUCTION comprises:

- an instruction to register REGISTER which processors are running at the beginning of the critical section;
- an instruction to shut down at least one processor (STOP_PROC) at the beginning of the critical section;
- an instruction to restart the processor (START_PROC) which was running at the end of the critical section.

Fig. 4A and Fig. 4B illustrate an example of interfering tasks. Fig. 4C and Fig. 4D illustrate how critical sections are defined according to the invention. In Fig. 4C, mutual exclusiveness of the critical section is implemented by means of a semaphore. In Fig. 4D, mutual exclusiveness of the critical section is implemented by disabling interrupts.

In Fig. 4A, two tasks T1 and T2 can access the same variables, specifically variable *x* representing an integer value and variable *even* representing a Boolean value. Variable *even* should indicate whether an even or an odd integer value is assigned to variable *x*. Task T1 assigns value '3' to variable *x* and subsequently value 'false' to variable *even*. Task T3 assigns value '6' to variable *x* and subsequently value 'true' to variable *even*. Tasks T1 and T2 can be executed simultaneously, as can be seen in Fig. 4B. The scheduler can interrupt task T1 after execution of the first statement, start the execution of T2 and resume the execution of T1 when T2 has terminated. The erroneous result is clear: the variable *x* has value '6' although the variable *even* indicates that variable *x* has an odd value.

Tasks T1 and T2 must be atomic actions to prevent this erroneous result. The programmer will define the complete source code of tasks T1 and T2 as critical sections using statements BCS() and ECS() for the beginning and the end of the critical section, respectively. In Fig. 4C, mutual exclusiveness of the critical section is implemented by a semaphore. In Fig. 4D, mutual exclusiveness of the critical section is implemented by disabling interrupts. The skilled person will recognize that a compilation system will usually perform transformations of the internal representation of the source code instead of transformations of the source code, since transformations of the internal representation are more efficient. The example pseudo-source code and pseudo-intermediate code are merely illustrative and do not describe a preferred way of inserting additional instructions into the source code. In most cases, additional instructions which are inserted into the source code are

preferably inserted into the internal representation of the source code, the additional instructions themselves also being formulated by the internal representation.

It is remarked that the scope of protection of the invention is not restricted to the embodiments described herein. Neither is the scope of protection of the invention restricted by the reference symbols in the claims. The word 'comprising' does not exclude other parts than those mentioned in a claim. The word 'a(n)' preceding an element does not exclude a plurality of those elements. Means forming part of the invention may both be implemented in the form of dedicated hardware or in the form of a programmed general-purpose processor. The invention resides in each new feature or combination of features.

CLAIMS:

1. A method (COMPILE) for translating source code (SC) into executable code (EC), characterized by the following steps:

- identify (IDENTIFY) the beginning and the end of at least one critical section of the source code (SC);
- determining (DETERMINE?) whether at least one additional instruction (INSTRUCTION) is needed to implement mutual exclusiveness of the critical section;
- if the additional instruction (INSTRUCTION) is needed, selecting (SELECT) the additional instruction (INSTRUCTION) and inserting (INSERT) the additional instruction (INSTRUCTION) into the source code (SC).

2. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises one or more explicit instructions.

3. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises a call upon a library routine.

4. A method (COMPILE) as claimed in claim 1, wherein the step of determining (DETERMINE?) comprises:

- checking whether task switching can occur during execution of the critical section;
- checking whether parallel tasks can be executed during execution of the critical section;
- deciding that no additional instruction (INSTRUCTION) is inserted into the source code (SC) if no task switching can occur and if no parallel tasks can be executed during execution of the critical section.

5. A method (COMPILE) as claimed in claim 1, wherein the step of determining (DETERMINE?) comprises:

- checking whether tasks can interfere during execution of the critical section;

- deciding that no additional instruction (INSTRUCTION) is inserted into the source code (SC) if no tasks can interfere during execution of the critical section.

6. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises:

- an instruction to save an interrupt flag (SAVE_INT) at the beginning of the critical section;
- an instruction to disable interrupts (DISABLE_INT) at the beginning of the critical section;
- an instruction to restore the interrupt flag (RESTORE_INT) at the end of the critical section.

7. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises:

- an instruction to acquire a semaphore (ACQUIRE_SEM) at the beginning of the critical section;
- an instruction to release the semaphore (RELEASE_SEM) at the end of the critical section.

8. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises:

- an instruction to register (REGISTER) which schedulers are running at the beginning of the critical section;
- an instruction to stop schedulers of interfering tasks (STOP_SCH) at the beginning of the critical section;
- an instruction to start the schedulers of interfering tasks (START_SCH) which were running, at the end of the critical section.

9. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises:

- an instruction to register (REGISTER) which schedulers are running at the beginning of the critical section;
- an instruction to stop all schedulers (STOP_ALLSCH) at the beginning of the critical section;

- an instruction to restart the schedulers (START_ALLSCH) which were running, at the end of the critical section.

10. A method (COMPILE) as claimed in claim 1, wherein the additional instruction (INSTRUCTION) comprises:

- an instruction to register REGISTER which processors are running at the beginning of the critical section;
- an instruction to shut down at least one processor (STOP_PROC) at the beginning of the critical section;
- an instruction to restart the processor (START_PROC) which was running, at the end of the critical section.

11. A compilation system arranged to translate (COMPILE) source code (SC) into executable code (EC), characterized in that the compiler is arranged to:

- identify (IDENTIFY) the beginning and the end of at least one critical section of the source code (SC);
- determine (DETERMINE?) whether an additional instruction (INSTRUCTION) is needed to implement mutual exclusiveness of the critical section;
- if the additional instruction (INSTRUCTION) is needed, select (SELECT) the additional instruction (INSTRUCTION) and insert (INSERT) the additional instruction (INSTRUCTION) into the source code (SC).

12. A compilation system as claimed in claim 11, wherein the beginning and the end of the critical section are identified (IDENTIFY) by predefined statements in the source code (SC).

13. A computer program product comprising a program, the program being conceived to be compiled by a compilation system as claimed in claim 12, wherein the program comprises at least one of the predefined statements.

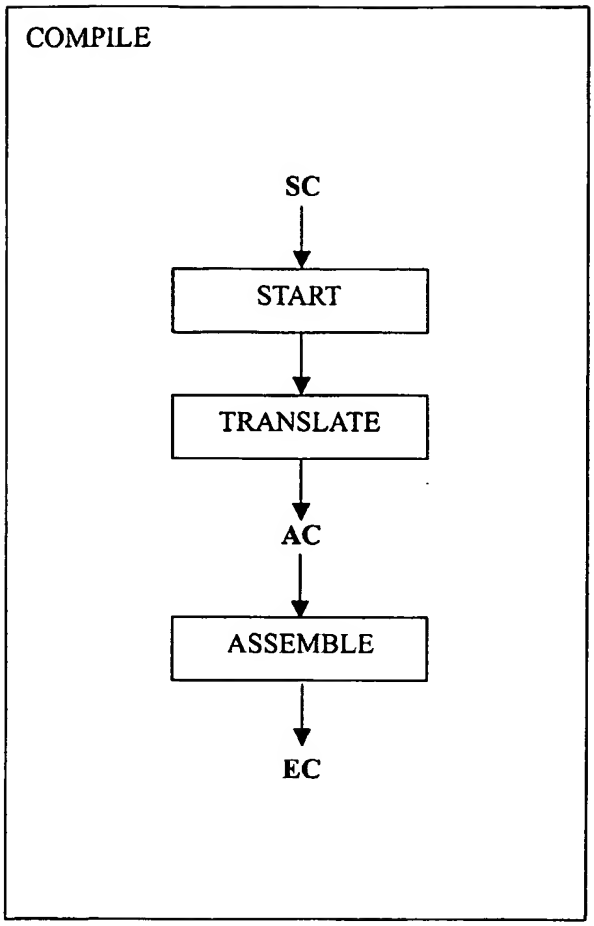


FIG.1

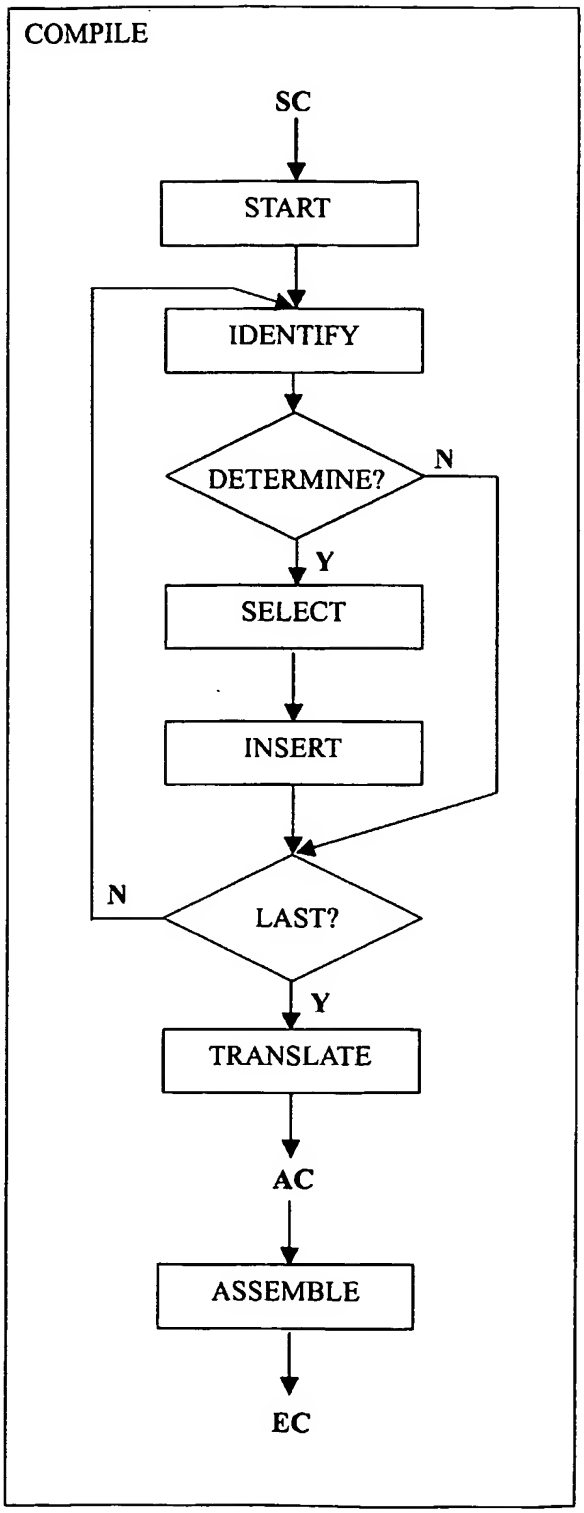

COMPILE
SC
START
IDENTIFY
DETERMINE?
N
Y
SELECT
INSERT
N
LAST?
Y
TRANSLATE
AC
ASSEMBLE
EC

FIG.2

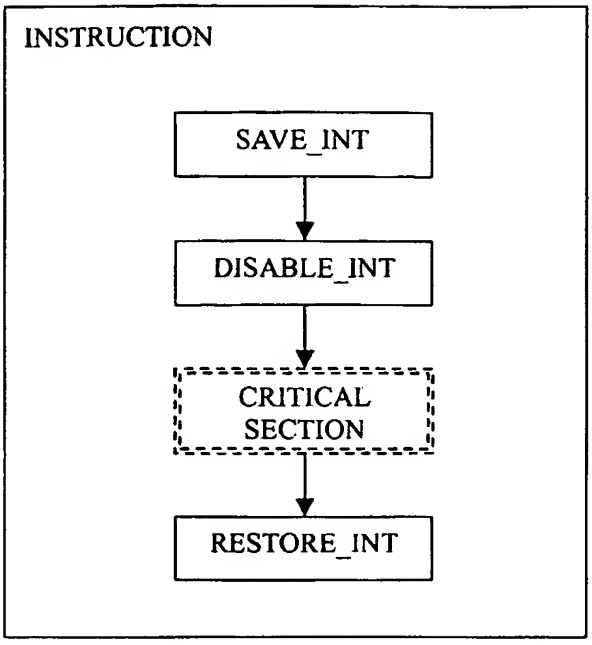

INSTRUCTION
SAVE_INT
DISABLE_INT
CRITICAL SECTION
RESTORE_INT

FIG.3A

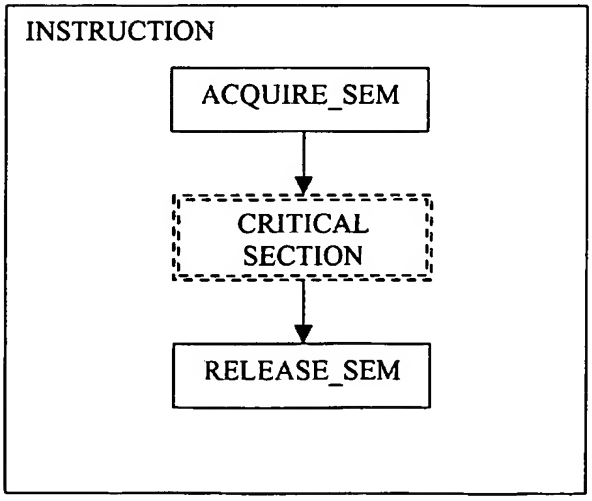

INSTRUCTION
ACQUIRE_SEM
CRITICAL SECTION
RELEASE_SEM

FIG.3B

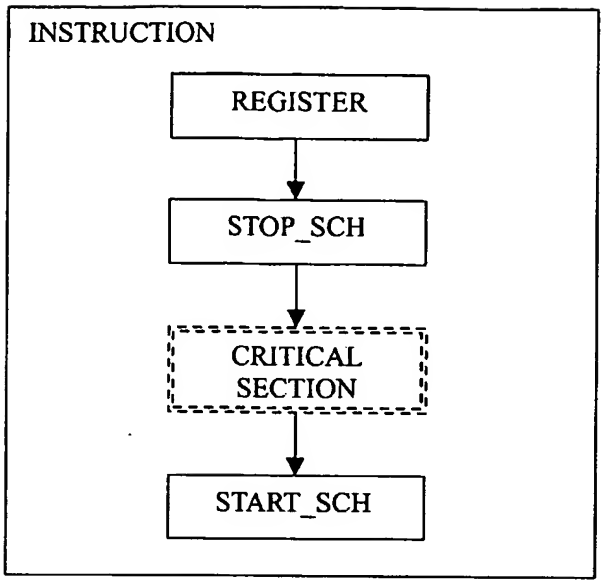

INSTRUCTION
REGISTER
STOP_SCH
CRITICAL SECTION
START_SCH

FIG.3C

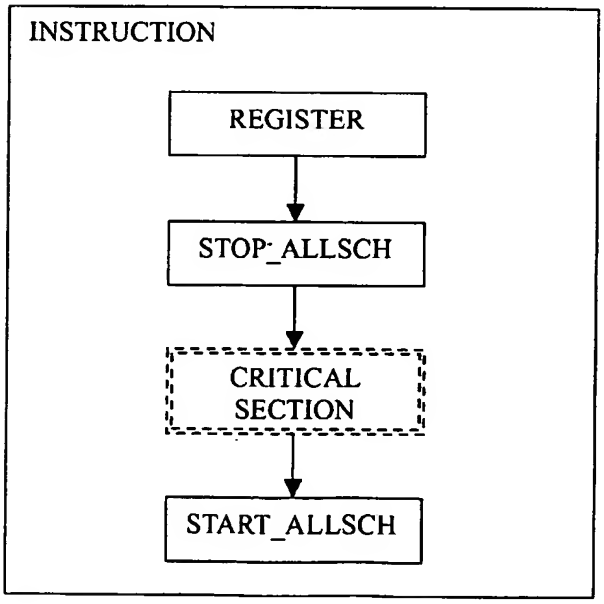

INSTRUCTION
REGISTER
STOP_ALLSCH
CRITICAL SECTION
START_ALLSCH

FIG.3D

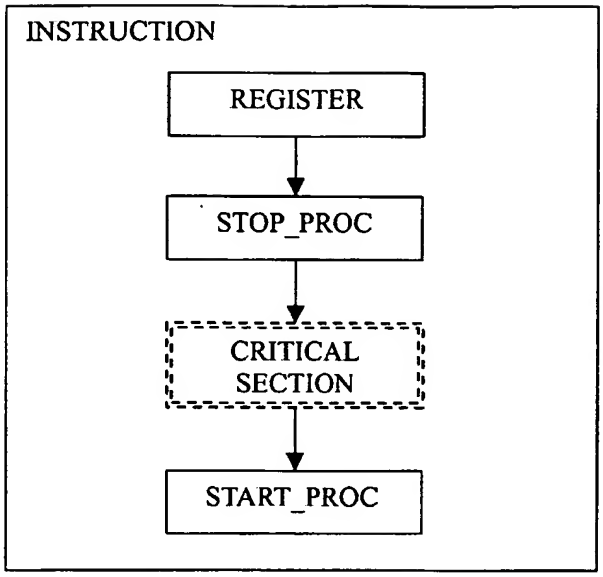

INSTRUCTION
REGISTER
STOP_PROC
CRITICAL SECTION
START_PROC

FIG.3E

int x; bool even;	

T1:
x = 3;
even = false;
	T2:
	x = 6;
	even = true;

FIG.4A

int x; bool even;	

T1:
x = 3;
.
.
even = false;
	T2:
	.
	x = 6;
	even = true;
	.

RESULT: x == 6 AND even == false	

FIG.4B

PSEUDO-SOURCE CODE	PSEUDO-INTERMEDIATE CODE

T1:
BCS();
x = 3;
even = false;
ECS();
	semAcquire(s);
	x = 3;
	even = false;
	semRelease(s);

T2:
BCS();
x = 6;
even = true;
ECS();
	semAcquire(s);
	x = 6;
	even = true;
	semRelease(s);

FIG.4C

PSEUDO-SOURCE CODE	PSEUDO-INTERMEDIATE CODE

T1:
BCS();
x = 3;
even = false;
ECS();
	save_flag = int_flag;
	int_flag = 0;
	x = 3;
	even = false;
	int_flag = save_flag;

T2:
BCS();
x = 6;
even = true;
ECS();
	save_flag = int_flag;
	int_flag = 0;
	x = 6;
	even = true;
	int_flag = save_flag;

FIG.4D